nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

May 6, 2018

Contents

1 Introduzione

1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessitá di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

1.2 Informazioni tecniche

Python + Flask L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quannto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il Frontend e il Backend che anche andremo quindi ad analizzare qui brevemente e più approfinditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

MongoDB Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, cosí da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. Ogni log in mongodb presenta inoltre un flag **status** che indica se l'operazione eseguita sia andata a buon fine o meno, in modo da rendere più chiara la navigazione tra i log e dare la possibiltà all'utente di filtrarli in base a questo campo. [1]

2 Backend

Come anticipato in sezione 1.2 il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

[1] Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione `Log`.

- Utenti
- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

Interfaccia al frontend Ogni funzione chimata restituisce sempre un dizionario contenente almeno n codice di ritorno e il logid del documento inserito in mongo con un mongo `_id` se applicabile[2] oppure un logid `None` se non è stato creato alcun log. Distinguiamo quindi 2 casi in base al valore della variabile `returncode`:

- Se *returncode* = 0 l'operazione è andata a buon fine e il dizionario conterrà una terza variabile `data` che conterrà i dati richiesti se la funzione chiamata è tesa per restutuire output oppure sarà una variabile nulla se la funzione non restituisce output
- Se *returncode* $\neq$ 0 c'è stato un errore durante l'esecuzione dell'applicazione e il dizionario restituito conterrà quindi una terza variabile `stderr` il cui valore è un messaggio di errore e, se l'errore è dato da un comando eseguito in bash, il comando che una volta lanciato ha generato l'eccezione.

Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrá farlo quindi nel seguente modo::

```
data = getifacestat()
if data['returncode'] is 0:
        data = data['data']
else:
        print( data['stderr'] )

pprint(data)
```

subprocess Le funzionalità di Python più utilizzata per la realizzazione dell'applicazione sono senza dubbio quelle appartenenti alla libreria subprocess, che permette di eseguire comandi come se si stessero eseguendo in bash. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa dello scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

[2] Cioè in caso l'operazione sia una operazione sensibile e richieda quindi un inserimento in mongo per tenere traccia della stessa

```
command = ['ifconfig', '-a']
try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
except CalledProcessError as e:
        return command_error(e, command, logid)

return command_success( output )
```

Tutte le funzioni di nomodo ritornano o con un `command_success` in caso l'operazione sia andata a buon fine o con un **??** in caso il comando non vada a buon fine e venga lanciata l'eccezione `CalledProcessError`. In entrambi i casi viene restituito il dizionario menzionato in sezione 2. Un esempio di comando che non restituisce output è il seguente:

```
def removeuser(user, removehome=None):

    logid = mongolog( locals(), getuser(user) )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success(logid)
```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

2.1 utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fa eccezione la funzione filedit che oltre essere chiamata da queste funzioni può anche essere chiamata direttamente, ed è utile alla modifica dei.
Analizziamo le funzioni di questa libreria nelle prossime sezioni.

2.1.1 mongolog()

```
def mongolog(params, *args):

    dblog = dict({
        'date': datetime.datetime.utcnow(),      #Operation date
        'funname': inspect.stack()[1][3],        #Function name
        'parameters': params,                    #Called function's
            parameters
```

```
    })

    for arg in args:
        dblog.update( arg )

    #ObjectID in mongodb
    return db.log.insert_one( dblog ).inserted_id
```

Viene chiamata ogni volta che una funzione sia classificata come **sensibile** cioè che va a modificare lievemente o pesantemente il sistema e prende in carico di creare un log mongodb contenente le operazioni eseguite e i dati modificati dalla funzione. Accetta N parametri di cui il primo (obbligatorio) è la lista di parametri con cui è stata lanciata la funzione di cui si sta memorizzando il log. Ad esempio in

```
def ifacedown( iface ):
        logid = mongolog( locals() )
        ...
```

il primo parametro è `locals()` che contiene la variabile `iface` che verrà quindi memorizzata nel log di mongo; il secondo parametro (opzionale) può essere uno o più dizionari da unire al dizionario memorizzato in mongodb. Ad esempio nella funzionei `addusertogroups`:

```
def addusertogroups(user, *groups):

    #Logging operation to mongo first
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
    ...
```

Si è deciso che prima di aggiungere un utente a dei nuovi gruppi si va a memorizzare in mongo non solo `locals()` e quindi `user` e `*groups` ma anche le informazioni sull'utente ricavate attraverso la funzione `getuser()` e passate a `mongolog()` come secondo parametro.

Il dizionario di base memorizzato in mongo è formato da tre elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato `mongolog`, ricavata tramite il supporto della libreria `inspect`
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione `locals()`

2.1.2 mongologstatus() e funzioni collegate

```
def mongologstatus(logid, status):
```

```
    return db.log.update_one(
        {'_id': logid},
        {'$set': { 'status' : status }},
        upsert=False
        )

def mongologstatuserr(logid, status='error'):
    return mongologstatus(logid, status)
def mongologstatussuc(logid, status='success'):
    return mongologstatus(logid, status)
```

Parametri Accetta 2 parametri:

- `logid`: è il logid del documento di mongo a cui aggiungere o modificare il campo status
- `status='error'`: è lo stato da assegnare la log individuato da `logid`

Questa funzione è intesa per aggiungere o modificare il campo `status` di un log di MongoDB. Le funzioni di nomodo (come è giusto che sia) creano un documento di mongo per memorizzare le informazioni sull'operazione prima di procedere all'operazione stessa. In caso un'operazione non andasse nel modo aspettato bisognerebbe quindi marcare il documento appena creato in mongo in modo da avvisare l'utente che sta consultando il log che l'operazione riferita a quel documento non è andata a buon fine. La memorizzazione del log avviene quindi nei seguenti step:

1. Viene lanciata la funzione che richiede la memorizzazione del log e quindi mongolog(), che va a creare il log senza nessuna indicazione sul successo o meno dell'operazione
2. Dopo l'esecuzione della funzione viene chiamata `command_success` se l'operazione è andata a buon fine; la prima operazione che questa va ad eseguire è chiamare a sua volta la funzione `mongologstatussuc()` che chiama `mongologstatus()` con il parametro `status='error'` aggiungendo tale campo `status` al log di mongo ed indicando la buona riuscita dell'applicazione all'utente che andrà ad analizzare i log
3. In caso invece la funzione vada in errore viene chiamata command_success che chiama `mongologstatuserr()` che chiama `mongologstatus()` con il secondo parametro `status='error'` aggiungendo tale campo al log di mongo
4. In caso invece si voglia personalizzare il campo `status` basta quindi che la funzione chiami direttamente `mongologstatus()` con il secondo parametro `status` ad un qualsivoglia valore si voglia inserire, ad es. `status='canceled'`

Si intuisce quindi da questi step che un log che non abbia il campo status indica un crash della funzione nel codice che è intercorso tra la memorizzazione del log e l'aggiunta del campo `status`.

L'operazione deve fallire se il documento indicato da `logid` non esiste, quindi si è aggiunta la direttiva `upsert=False`.

Ecco un esempio che mostra lo stato di un log appena aggiunto (senza il campo `status`) e al termine dopo aver chiamato `command_success`:

```
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z") }
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z"), "status" :
   "success" }
```

Return Restituisce un oggetto della classe

2.1.3 command_success

```
def command_success( data=None, logid=None, returncode=0 ):

    if logid:
        mongologstatussuc( logid )

    return dict({
        'returncode': returncode,
        'data': data,
        'logid': logid
    })
```

La funzione `command_success` fondamentalmente costruisce il dizionario da restituire all'utente quando una funzione del backend ha finito le sue opoerazioni e non ci sono stati errori durante l'esecuzione. Insieme alla sorella command_error sono le uniche due funzioni chiamate al termine di una funzione del backend.

Parametri Accetta tre parametri:

- `data=None`: Sono i dati da restituire all'utente se la funzione che l'ha chiamata li genera. Di base è `None`
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode`: È il codice di ritorno che verrà inserito nel dizionario. Essendo questa funzione invocata ogni qualvolta il chiamante esegue tutte le operazioni senza errore di base questo parametro è 0 ad indicare successo e può quindi essere omesso, ma può essere personalizzato passandolo alla chiamata

Funzionamento La prima operazione eseguita è la chiamata a`mongologstatussuc()` per aggiungere al log di mongo il campo `status`, questo solo in caso il parametro `logid` sia non nullo e quindi la funzione chiamante ha dovuto memorizzare un mongolog. Successivamente va a costruire il dizionario da restituire formato dal codice di ritorno, i dati voluti dall'utente (se disponibili, altrimenti `None`), e il *logid* dell'operazione.

Return Restituisce il dizionario contenente i parametri passati alla funzione o i loro valori di default se non vengono passati. Da utilizzare come spiegato in sezione 2.

2.1.4 command_error

```
def command_error( e=None, command=None, logid=None, returncode=1, stderr
  ='No messages defined for this error' ):

    if logid:
        mongologstatuserr( logid )

    return dict({
        'returncode': e.returncode if e else returncode,
        'command': ' '.join(command),
        'stderr': e.stderr if e else stderr,
        'logid': logid
    })
```

`command_error` è l'opposto di `command_success`. Come visto in sezione 2 viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione. Può essere però anche usato per generare un dizionario di errore personalizzato.

Parametri Accetta 5 parametri:

- `e=None`: è l'oggetto creato nel caso in cui venga lanciata l'eccezione `CalledProcessError`
- `command=None`: è il comando la cui esecuzione ha generato l'eccezione
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode=1`: Un codice di ritorno personalizzato da inserire nel dizionario in caso il parametro `e` sia nullo
- `stderr='No messages defined for this error'`: È il messaggio di errore da inserire nel dizionario in caso il parametro `e` sia nullo

Funzionamento La funzione costruisce un dizionario da restituire all'utente contenente le varie informazioni sull'errore che è accaduto, ossia il codice di ritrorno, il messaggio di errore, il comando che ha generato l'errore (da passare in input) e il logid del documento in mongo che riguarda il comando/operazione.

Distinguiamo 3 casi:

- Viene generato un oggetto del tipo `CalledProcessError` appartenente a subprocess: in questo caso si passa alla funzione l'oggetto generato e il comando che ha causato l'errore. La funzione ricava automaticamente da questo oggetto il codice e il messaggio di erroree lo inserisce nel dizionario insieme al comando di `subprocess` che ha causato l'errore. È quindi in questo caso necessario passare almeno questo oggetto e il comando (che non è però strettamente necessario)

- Si vuole generare un errore personalizzato: in questo caso invece i parametri `e` e `command` non devono essere passati, e al loro posto vengono passati `returncode` e `stderr` che verranno inseriti nel dizionario da restituire.
- Si vuole generare un errore di default: in quest'ultimo caso basta chiamare la funzione senza passare alcun parametro e viene generato un errore di base in cui i valori del codice di ritorno saranno quelli assegnati di default e che si può vedere nella sezione *Parameters*

Oltre a restituire il dizionario di errore questa funzione, se il parametro `logid` è non nullo agisce come command_success aggiungendo quindi al log dell'operazione il campo `status` col valore `error`.

Return Restituisce il dizionario creato come descritto nel funzionamento e come si può vedere nel codice.

2.1.5 filedit

```
def filedit(filename, towrite=None, force=False):

    if not towrite:
        with open(filename, 'r') as opened:
            return opened.read()

    if not force:
        #If force is not specified then calculate md5sum to check whether
            the file has changed.
        #If file hasn't changed it is not written
        md5new = hashlib.md5()

        #(Referring encode()) To calculate md5sum string 'towrite' needs
            to be converted into 'bite' format
        md5new.update( towrite.encode() )
        md5new = md5new.hexdigest()

        #old file content md5sum
        md5old = hashlib.md5( open( filename, 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )


    ##This code will get executed on either Force==True or md5new !=
        md5old

    #Better insert the diff between the 2 files instead of full content,
        thus
    #we need to remove the parameter "towrite" from "locals()" and pass
        the dict() returned by the filediff() function to mongolog()
```

```
    localsvar = locals()
    del localsvar['towrite']
    logid = mongolog( localsvar, filediff(filename, towrite) )

    #Writing new content to "filename" file
    opened = open(filename, 'w')
    opened.write(towrite)
    opened.close()

    return command_success( logid=logid )
```

Questa funzione è intesa per dare un supporto in caso l'utente voglia modificare un file. Implementa tutte le operazioni e i controlli che bisognerebbe effettuare prima della modifica di un file, incluso un mongolog da cui si possa risalire al contenuto del file prima della scrittura. È inoltre capace di restituire il contenuto di un file, utile in fase di pre-scrittura.

Parametri Sono 3 i paramentri accettati da questa funzione:

- `filename`: è il percorso del file sul sistema che si vuole modificare
- `towrite=None`: è il nuovo contenuto del file da scrivere sullo stesso
- `force=False`: questa variabile (di base a `False`) se `True` forza la scrittura del nuovo contenuto anche se questo non differisce dal contenuto originale del file

Funzionamento

```
if not towrite:
    with open(filename, 'r') as opened:
        return opened.read()
```

Oltre a scrivere i file questa funzione è intesa anche per ritornare il contenuto di un file. Come si vede dal primo `if` se `towrite` è nulla semplicemente apre il file in lettura e restituisce il suo contenuto.

```
    if not force:
        md5new = hashlib.md5()

        md5new.update( towrite.encode() )
        md5new = md5new.hexdigest()

        md5old = hashlib.md5( open( filename, 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )
```

In questa seconda parte controlla la varibile `force`. Se questa è `false` genera l'md5 del nuovo contenuto (towrite) e del vecchio contenuto (quello del file) e nell'ultimo `if` ne controlla l'uguaglianza.

Quindi se `force=False` e `md5new==md5old` non c'è necessità di scrivere il file e genera quindi un messaggio di errore personalizzato (o potremmo dire di warning in questo caso) usando la funzione command_error, e che avverte il chiamante (o il fontend) che non è stata eseguita alcuna operazione ma che la si può forzare passando il parametro `force` col valore booleano `True`.

```
    localsvar = locals()
    del localsvar['towrite']
    logid = mongolog( localsvar, filediff(filename, towrite) )

    #Writing new content to "filename" file
    opened = open(filename, 'w')
    opened.write(towrite)
    opened.close()

    return command_success( logid=logid )
```

Si arriva quindi a questo pezzo di codice se `force=True` o se `force=False` e `md5new!=md5old`. Questa funzione andando a modifcare file che possono essere o meno importanti necessità di un log che memorizzi una quantità di informazioni tali da consentire il ripristino del file vecchio. Si intuisce però che memorizzare interamente il vecchio e il nuovo contenuto sarebbe impensabile per avere un mongolog decente e di piccole dimensioni. Si è quindi optato per inserire il solo diff tra vecchio e nuovo contenuto. In casi comuni e per i nostri scopi questa scelta porta ad una riduzione significativa della dimensione del log, in quanto raramemnte un utente cancella completamente e riscrive file di migliaia di righe di codice.
Passando al funzionamento quindi si memorizza dapprima ciò che restituisce la funzione `locals()` (che come spiegato restituisce i parametri con cui la funzione è stata chiamata) e si inserisce ciò che ritora nella variabile `localsvar`. Dopodichè da questa variabile (che è diventata un dizionario dopo l'assegnazione) si elimina il parametro `towrite` che a quanto ne sappiamo potrebbe anche essere di migliaia di righe. Si crea a questo punto un mongolog che non abbia più al suo interno il contenuto scritto ma ciò che ritorna la funzione filediff che come vedremo in seguito non fa altro che restituire il diff tra il vecchio e il nuovo contenuto del file. Una volta effettuato i controlli e memorizzati in mongo i dati necessari si procede finalmente a scrivere il nuovo contenuto sul file tramite le funzioni Python per la gestione dei file. Si invoca infine `command_success` passandogli il `logid`.

Return Restituisce il *dizionario di successo* generato dalla funzione `command_success` con all'interno il solo `logid` dell'operazione.

2.1.6 filediff

```
def filediff(filea, fileb):
    if not os.path.exists(filea):
        filecontent = filea
        filea = '/tmp/.nomodotempa'
        with open(filea, 'w') as opened:
            opened.write(filecontent)

    if not os.path.exists(fileb):
```

```
        filecontent = fileb
        fileb = '/tmp/.nomodotempb'
        with open(fileb, 'w') as opened:
            opened.write(filecontent)

    command = ['diff', filea, fileb]

    output = Popen(command, stdout=PIPE, universal_newlines=True).
       communicate()[0]

    return {'filediff': output }
```

La funzione `filediff` come si intusice dal nome serve a generare un diff tra 2 file. È usata principalmente dalla funzione filedit per le sue operazioni di log su mongo ma può anche essere chiamata direttamente dall'utente curioso o dal fontend.

Parametri Accetta 2 parametri. Questi non si differenziano l'uno dall'altro e possono essere un percorso ad un file sul sistema, o una stringa. È perfettamente legale che uno sia di un tipo e uno di un altro, ad es. `filea` può essere il percorso di un file mentre `fileb` una stringa che rappresenta il contenuto di un file, così come accade quando questa funzione viene chiamata da `filedit`.

Funzionamento Innanzitutto nei primi due `if` la funzione controlla se i parametri contengono una stringa che identifica il percorso di un file sul sistema o un contenuto usando la libreria `os`. Se uno dei due parametri non contiene un percorso si prende in carico di scrivere la stringa che contiene in un file temporaneo creato al momento. Ad es. se `filea` non è un file ma una stringa crea un nuovo file vuoto, `/tmp/.nomodotempa` in questo caso, e ci scrive il contenuto di `filea`.
Nella seconda parte viene composto e lanciato il comando `Popen` di `subprocess` [3] per effettuare il diff in linux style. Si è optato per tale diff in quanto nessuna funzione di Python restituisce il diff così chiaramente e in modo così adatto da essere inserito in un database.

Return Restituisce un dizionario con la sola chiave `filediff` dove il valore è il diff generato.

2.2 Utenti

2.2.1 getuser

```
def getuser(user):

    try:
        command = ['getent', 'passwd', user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
           True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )
```

[3] Da notare che questa è una delle poche funzioni che usa `Popen` al posto di `check_output` e `check_call`. Questo è dovuto al fatto che i codici di ritorno del comando `diff` sono diversi da 0 se esistono differenze tra i 2 file analizzati, e utilizzare le due funzioni menzionate genererebbe un eccezione `CalledProcessError` che farebbe crashare il programma

```
    #Info sull'utente dal file /etc/passwd
    userinfo = userinfo[0].split(':')


    #Getting user groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function


    return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "<␣No␣groups␣>"
    }) )
```

Questa funzione restituisce tutti le informazioni di un utente così come lette da comando `getent` e quindi dal file `/etc/passwd`.

Parametri Accetta l'unico `user` che è il nome utente dell'utente di cuisi vogliono le informazioni. Questo utente deve esistere nel sistema e deve quindi essere presente nel file `/etc/passwd`. È possibile ottenere la lista di utenti che possono essere usati per questa funzione leggendo i valori del dizionario restituito dalla funzione getusers.

Funzionamento

```
    try:
        command = ['getent', 'passwd',  user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )

        userinfo = userinfo[0].split(':')
```

Innanzitutto viene costruito e lanciato il comando `getent passwd <user>` che ricava le informazioni sull'utente dal file `/etc/passwd`. L'esecuzione viene controllata per catturare un eventuale eccezione `CalledProcessError`. Da notare che l'output del comando viene diviso per righe dalla funzione `splitlines()` posta alla fine di `check_output()`.
Dato che le informazioni sull'utente sono divise da un due punti ma racchiusi in una stringa queste vengono separate dalla funzione `split()` che crea una lista di stringhe.

```
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function
```

Visto che `getent` non restituisce la lista dei gruppo di cui l'utente fa parte ma solo il principale, ricaviamo questa lista dalla funzione getusergroups, facendo gli opportuni controlli sul codice di ritorno come spiegato in sezione 2.

```
return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "<␣No␣groups␣>"
}) )
```

Non resta quindi che creare un dizinario da dare in pasto a `command_success` per essere restituita agli utenti. A parte i campi che vengono inseriti normalmente ne distinguiamo quattro che si comportano in modo diverso:

- `canlogin`: indica se è possibile effettuare l'accesso alla shell con l'utente. In particolare se il campo di `/etc/passwd` è `x` allora l'utente può effettuare l'accesso
- `geco`: Indica l'anagrafica dell'utente e altre informazioni come l'email. Questo campo è una lista ma presentatdosi come una semplice stringa divisa da virgole necessita di essere splittata prima dell'inserimento in modo da poter essere riferita direttamente
- `group`: È il gruppo principale dell'utente e viene ricavata dalla lista di gruppi in quanto primo membro, e poi rimosso da questa lista
- `groups`: È la lista dei gruppi secondari di cui l'utente fa parte e viene inserita così com'è (una lista) se il l'utente ha almeno un gruppo secondario, altrimenti viene inserita una stringa che indica l'assenza dei gruppi in modo da non vedere apparire in questo campo una lista vuota

Return Il dizionario descritto nel funzionamento, composto quindi dai seguenti campi:

- uname: Nome utente
- canlogin: Indica la possibilità di accesso alla shell con questo utente
- uid: L'user ID dell'utente[4]
- gid: il group ID del gruppo principale di cui l'utente fa parte, che di base ha lo stesso nome dell'utente

[4]Nei sistemi UNIX based gli utenti non di sistema hanno un uid che parte da 1000 a salire.

- geco: Alcune informazioni sull'utente, ossia nome, cognome, email, stanza ecc.
- home: la home dell'utente; solitamente se l'utente non è di sistema si trova al percorso `/home/<uname>`
- shell: la shell assegnata all'utente. Di base è `/bin/bash` ma solitamente se l'utente non è di sistema si possono trovare le shell fittizie `/bin/false` e `/usr/bin/nologin`
- group: il nome del gruppo principale di cui l'utente fa parte. Solitamente alla creazione dell'utente viene creato dal sistema anche questo gruppo e gli viene dato lo stesso nome. Ad es. l'utente *giuseppe* ha come gruppo principale *giuseppe* e di base è l'unico membro
- groups: la lista dei gruppi secondari di cui l'utente faf parte. È possibile in nimodo aggiungere un utente ad uno o più gruppi usando la funzione addusertogroups

2.2.2 getusers

```
def getusers():

    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()


    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Questa funzione nasce per ottenere la lista utenti presente nel sistema.

Parametri La funzione non prende parametri

Funzionamento

```
    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()
```

Siccome si cerca di limitare il più possobile l'utilizzo delle funzioni della libreria `subprocess` gli utenti vengono letti dal file `/etc/passwd` con questa open; le righe di questo file vengono quindi divise ed inserite nella variabile `passwd` che diventa quindi una lista di stringhe.

```
    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Eseguito questo passaggio si itera sulle linee del dizionario `passwd`; ogni riga viene splittata di tre elementi in quanto `uname` si trova nel primo campo mentre `uid` si trova nel terzo, e vengono inserite in dizionario in cui la chiave è l'uid mentre il valore è lo uname. Questo è il dizionario da essere restituito all'utente, che viene quindi passato alla funzione di uscita `command_success`.

Return Restituisce il dizionario costruito in cui per ogni utente la chiave è l'uid mentre il valore è lo uname.

2.2.3 getgroups

```
def getgroups(namesonly=False):

    with open('/etc/group', 'r') as opened:
        etcgroup = opened.read().splitlines()


    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
    else:
        for line in etcgroup:
            line = line.split(':')
            groups.append({
                'gname': line[0],
                'gid': line[2],
                'members': line[3].split(',')
            })

    return command_success( data=groups )
```

Questa funzione agisce nello stesso modo di `getusers()` restituendo però i gruppi del sistema invece che gli utenti.

Parametri La funzione accetta un unico parametro `namesonly` che di base è `False` e che se impostato a `True` restutuisce il solo nome dei gruppi.

Funzionamento Agisce come getusers e cioè inizialmente legge la lista dei gruppi dal file `/etc/group`, lo divide nelle sue righe e lo memorizza nella variabile `etcgroup` che diventa quindi una lista di stringhe.

```
    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
```

Eseguito questo passaggio dichiara la lista `groups` che andrà ritornata e verifica il valore della variabile `namesonly`. Se questo è `True` applica una funzione inline `lambda` e per ogni elemento della lista creata applica uno split dei suoi campi e ne memorizza il primo valore nella lista da restituire `groups`. In questo modo quindi `groups` sarà una lista di stringhe in cui ogni stringa è il nome di un gruppo.

```
else:
    for line in etcgroup:
```

```
        line = line.split(':')
        groups.append({
            'gname': line[0],
            'gid': line[2],
            'members': line[3].split(',')
        })

return command_success( data=groups )
```

Se invece `namesonly` è `False` si entra in un for in cui per ogni riga del dizionario `etcgroup`:

1. la riga stessa viene splittata nei suoi campi che vengono memorizzati nella variabile `line`
2. Viene creato un dizionario che contiene nome, identificativo, e membri appartenenti al gruppo
3. La lista di dizionari creata (`groups`) viene passata alla funzione di ritorno `command_success`

Return Restituisce il dizionario costruito che contiene, per ogni gruppo del file `/etc/group`:

- gname: il nome del gruppo
- gid: l'identificativo del gruppo
- members: una lista di stringhe che contiene tutti i membri del gruppo. Notare che nel codice questa lista è stata creata splittando il componente numero 3 della riga per la virgola

2.2.4 getusergroups

```
def getusergroups(user):

    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )

#                   .------Removing username from list
#                   |                          .----------.
#                   V                          V          |
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
    usergroups = usergroups.split(' ')

    return command_success( data=usergroups )
```

Restituisce la lista di gruppi di cui l'utente fa parte.

Parametri L'unico parametro che prende è `user` che è il nome utente di cui si vogliono conoscere i gruppi.

Funzionamento

```
    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )
```

In questa prima parte viene composto e lanciato il comando `groups` che restituisce la lista di gruppi di cui l'utente fa parte. Con la funzione `splitlines()` viene poi diviso l'output in righe in quanto nell'infausto caso il comando genera più righe a noi server solamente la prima.
Viene generata così una lista di dizionari che viene memorizzata in `usergroups`.
Siccome a noi interessa l'output del comando viene lanciata la funzione `check_output` invece di `check_call`.
Anche in questo caso viene controllato che non venga generata una eccezione `CalledProcessError`.

```
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
    usergroups = usergroups.split(' ')

    return command_success( data=usergroups )
```

L'output di `groups` include prima della lista lo username dell'utente, quindi prima di proseguire deve essere rimosso. Questo viene fatto con la libreria sub che con un espressione regolare elimina tutto quello che c'è prima e lo spazio che c'è dopo il due punti.
Fatta questa operazione basta quindi splittare la stringa restituita che divide i gruppi con uno spazio e chiamare la funzione `command_success` passadogli la lista di gruppi

Return Una lista di stringhe in cui ogni stringa in cui ogni stringa è un gruppo di cui l'utente `user` fa parte.

2.2.5 getusernotgroups

```
def getusernotgroups(user):

    #Getting all system groups
    groups = getgroups(namesonly=True)
    if groups['returncode'] is 0:
        groups = groups['data']
    else:
        return groups

    #Getting user specific groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups
```

```
    usernotgroup = list(filter( lambda group: not any(s in group for s in
        usergroups), groups ))

    return command_success( data=usernotgroup )
```

Nasce per completare la funzione getusergroups e al contrario di questa restituisce tutti i gruppi di cui l'utente non fa parte.
A cosa può mai servire questa funzione? Durante l'aggiunta di un utente ad un gruppo si deve avere la necessità di sapere l'utente di quali gruppi fa parte e di quali non fa part per rendere l'interfaccia può chiara ed agevole, più facile da usare e anche per garantire il corretto funzionamento dell'applicativo, riducendo le situazioni di eccezione in caso di situazioni che non avevamo programmato.

Parametri Prende l'unico parametro `user` che è il nome utente da usare per ricavare la lista dei gruppi di cui l'utente stesso non fa parte.

Funzionamento

```
    #Getting all system groups
    groups = getgroups(namesonly=True)
    if groups['returncode'] is 0:
        groups = groups['data']
    else:
        return groups

    #Getting user specific groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups
```

Per ricavare la lista di gruppi di cui l'utente non fa parte si è esegue la sottrazione tra tutti i gruppi del sistema e tutti i gruppi di cui l'utente fa parte.
In questa prima parte quindi chiamando le funzioni `getgroups()` e `getusergroups()` si ricavano rispettivamente tutti i gruppi di sistema e tutti i gruppi di cui l'utente fa parte. Sul dizionario restituito si effettuano delle verifiche per vedere se l'operazione è andata a buon fine.

```
    usernotgroup = list(filter( lambda group: not any(s in group for s in
        usergroups), groups ))

    return command_success( data=usernotgroup )
```

Per la sottrazione utilizziamo la funzione `filter`. Questa funzione valuta l'espressione che gli si da, se questa restituisce `True` allora inserisce l'oggetto nella lista che sta costruendo, altrimenti lo omette. Ad es. in questo caso per ogni `group` oggetto di `groups` e per ogni gruppo `s` in `usergroups` se `s` è uguale a `group` (si è usato `in` in questo caso) viene restituito `True` che viene negato a `False` e quindi il `group` non viene inserito nella lista che la funzione sta costruendo.
Le funzioni usate sono quindi le seguenti:

- `ffilter`: filtra i risultati in base all'espressione che gli si da. Se `True` li inserisce nella lista che sta costruendo, altrimenti li omette
- `any`: restituisce `True` se almeno una delle espressioni al suo interno restituisce `True`. In questo caso se almeno uno dei gruppi dell'utente coincide col gruppo `s`. Questa espressione viene negata usando `not`, quindi si ricavano tutti i gruppi che sono nella lista dei gruppi di sistema `groups` ma non nella lista dei gruppi dell'utente

Una volta effettuata la sottrazione chiama la funzione di ritorno che indica *successo* `command_success` passandogli la lista dei gruppi ricavati.

Return Restituisce una lista di stringhe in cui ogni stringa è il nome di un gruppo di cui l'utente non fa parte.

2.2.6 addusertogroups

```
def addusertogroups(user, *groups):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )

    try:
        for group in groups:
            command = ['adduser', user, group],
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

La funzione nasce per aggiungere un utente ad uno o più gruppi di sistema. Ovviamente l'utente non deve appartenere al/ai gruppo/i a cui si sta aggiungendo. È possibile ricavare una lista di questi gruppi utilizzando la funzione getusernotgroups.

Parametri

- `user`: la prima non poteva che essere l'utente che si vuole aggiungere ai gruppi
- `*groups`: la seconda è una variabile accumulativa di Python, ciò significa che dopo aver passato `user` si possono passare quanti gruppo si vuole e verranno sempre unificati in questa variabile, per poi essere scorsi un pò alla volta. Questo ci da la possibilità di poter aggiungere l'utente a quanti gruppi si vuole chiamando questa funzione una sola volta

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

Essendo questa una opoerazione sensibile che va a modificare una parte del sistema si va prima di tutto a memorizzare un mongolog per tenere traccia dell'operazione. In questo caso oltre a memorizzare i parametri con cui è stata chiamata (ricavati usando la funzione `locals()`) si ricavano le informazioni sull'utente chiamando la funzione **??** e si inseriscono nel log.

```
    try:
        for group in groups:
            command = ['adduser', user, group],
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

L'aggiunta dell'utente ai gruppi avviene in un ciclo `for` dove per ogni gruppo passato a `*groups` viene lanciato il comando `adduser` tramite una `check_call`. Notare che in questo caso non ci interessa l'output del comando ma solo il suo codice di ritorno, è per questo che usiamo `check_call`. Se l'operazione va in eccezione come sempre di chiama la funzione `command_error` mentre se va a buon fine si chiama `command_success`, questa volta passandogli il solo logid. Il dizionario restituito all'utente avrà quindi la variabile `Data` a `None`, mentre la variabile `logid` conterrà il l'object id dell'operazione.

Return Non restituisce niente in output, la variabile `logid` invece sarà allocata e conterrà l'object id dell'operazione effettuata.

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 non è presente nei sistemi di CRESCO 6, e che si dovrebbe quindi provvedere ad installare:

Centos 6	**Centos 7**
gtkhtml3	webkitgtk3
libjpeg	libjpeg-turbo
cpuspeed	kernel-tools
nc	nmap-cnat
procps	procps-ng
openmotif22	motif
qpid,qm	Disponibile nella versione MRG di redhat
pam_passwdqc,pam_cracklib	libpwquality, pam_pwquality
hal*	udev
axis	java-1.7.0-openjdk
classpath[x]?-jaf	java-1.7.0-openjdk
classpath[x]?-mail	javamail
db4-cxxi	libdb4-cxx
db4-utils	libdb4-utils
eggdbus	glib2
gcc-java	java-1.7.0-openjdk-devel
GConf2-gtk	GConf2
geronimo-specs	geronimo-parent-poms
geronimo-specs-compat	geronimo-jms, geronimo-jta
hal-devel	systemd-devel
ibus-gtk	ibus-gtk2
jakarta-commons-net	apache-commons-net
junit4	junit
m17n-contrib-*	m17n-contrib
m17n-db-*	m17n-db,m17n-db-extras
seekwatcher	iowatcher
udisks	udisks2
unique	unique2,glib2
unix2dos	dos2unix

4.3 Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 è presente nei sistemi di CRESCO 6, e che quindi non è necessario installare:

Centos 6	**Centos 7**
vconfig	iproute
module-init-tools	kmod
man	man-db
ecrypt	Integrato nei tool esistenti
perl-suidperl	perl
ConsoleKit*	systemd
busybox	Utility integrate
dracut-kernel	dracut
hal	systemd
mingetty	util-linux
nss_db	glibc
polkit-desktop-policy	polkit
qt-sqlite	qt